# A PLOTTING TOOL FOR INTERNET BASED ON CLIENT/SERVER COMPUTING MODEL

MENG CAI

A MAJOR REPORT

IN

THE DEPARTMENT

OF

COMPUTER SCIENCE

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF MASTER OF COMPUTER SCIENCE

CONCORDIA UNIVERSITY

MONTREAL, QUEBEC, CANADA

SEPTEMBER 2001

0-612-64076-0

# ABSTRACT

## A Plotting Tool for Internet Based on Client/Server Computing Model

Meng Cai

This report outlines the design, implementation, and deployment of a plotting tool based on client/server distributed computing model. The tool plots mathematical functions for a remote user over Internet. It provides a complete and friendly graphical user interface and can be loaded into a conventional Web browser. The tool is simple, cheap, easy to learn and easy to use. It is also reliable, portable and extensible.

The tool is designed and implemented based on the component programming model, and integrates many cutting edge technologies, including Java Remote Method Invocation (RMI), Java Native Interface (JNI), Java Foundation Class (JFC), and Java Plug-in. Overviews of these technologies are given. For some of them, the alternative technologies are also discussed, and the reasoning for selecting a particular technology is presented.

In addition, this report demonstrates the steps to reuse a legacy module written in C++ in the plotting tool developed in Java. The benefits of reusing this module are discussed.

## Acknowledgements

I would like to express my sincere gratitude to my supervisor, Professor Peter Grogono, for his guidance, support, patience and encouragement during the entire course of this work.

# Contents

# List of Figures

# Chapter 1

# Introduction

In this chapter, I will briefly discuss the motivation for developing a 2-D plotting utility based on the client/server model. The goals and the functional requirements of this utility will also be summarized.

## 1.1 Project motivation

A 2-D plotting utility is very useful in helping people to visualize a mathematical function or compare a group of functions. Today, there are plenty of plotting tools in the market, for example, *Mathematica* [1], *MatLab* [2], and *Maple*[3]. Unfortunately, most of them are expensive, hard to learn, and too sophisticated for casual users.

Recently, Ahn Phong Tran developed a simple 2-D graph plotter [4]. The plotter is reliable, efficient and very easy to use. However, Tran's plotter is a standalone application, and as a result, every user has to install a copy of it on his/her own computer. In addition, when updating the software, a new version has to be distributed to each user and the user has to re-install it.

On the other hand, rapid developments in computer networks and World Wide Web make it feasible to let many users share a single copy of an application. A graph plotting utility based on client/server distributed computing model perfectly satisfies this need. With the client/server paradigm, users may load the client component using a Web browser across an organization's Intranet or across the Internet, and access remotely the server component installed on a host computer. Hence, a user only needs to have a conventional browser to use the plotting utility. In this way, we may save the computing resources and it is also particularly convenient for occasional users. Furthermore, once the application is updated on the server computer, a user will automatically load the latest version of the client component and access the latest version of the server.

## 1.2 A Summary of the Previous Work

Before describing the goals and the requirements of this project, let's summarize the previous work completed by Ahn Phong Tran [4].

What Tran accomplished is a standalone 2-D Graph Plotter application which can be used to plot simple algebraic expressions. At the start of the application, an *Input Dialog* is displayed to let user enter an expression. Next, user clicks the "Checking Expression Syntax" button on the dialog to parse the expression based on the following grammar.

*Expr -> ['-'] Term {( '+' | '-' ) Term}.*

*Term -> Factor {('*' | '/' ) Factor}.*

*Factor -> Primary ['^' Primary].*

*Primary -> NUM | VAR [ '('Expr') ' ] | '('Expr')'.*

Where *NUM* represents a number, *VAR* represents either a simple variable or a built-in function (for example, *sin*, *cos*, *tan*, *atan*, *exp*, *log*, *fabs*, or *sqrt*). Things of the form [X] means that X may appear or not. Things of the form (X1|X2) indicates that one of the Xi must appear once. Things of the form {X} indicate that X can occur zero or more times.

If the expression is not syntactically correct, an error is reported. Otherwise, on the same dialog, the user enters a variable name and its range, and optionally, a parameter name, its step and range, and some constants and their values. The user then clicks the "Plot" button, and a number of points are generated and plotted on the screen.

Internally, Tran's application consists of two major components, namely, a *parser/evaluator* component and a *user interface* (including *plotting*) component.

The *parser/evaluator* component is responsible for parsing an expression, setting the variable values in the expression, and gives back a value of the expression. To accomplish this, it checks the expression syntactically, builds a parsing tree if the expression is syntactically correct, assigns values to the nodes in the parsing tree, and returns the value of the whole parsing tree. The parser was based on the *recursive descent method* [5] and was written in C++ programming language.

The *user interface* component provides an *Input* dialog for user inputs and a *View* on which curves can be drawn. This component was implemented with *Microsoft Foundation Class* (MFC). The *Input* dialog inherits MFC's *CDialog* class, and the View derives from MFC's *CView* class.

Tran's tool is simple, easy to learn and it works well. However, the fact that it was implemented as a standalone application makes it inconvenient in some circumstances, as mentioned in Section 1.1.

## 1.3 Project goals

The goal of this project is to extend the above 2-D graph plotting tool based on the distributed computing paradigm mentioned in Section 1.1. The application needs to be component-oriented. More specifically, we need to develop a client component that interacts with a user, and a server component that is responsible for expression parsing and evaluation. The client and server components should be relatively independent, but the client must be able to communicate with the server component via a well-defined remote interface. The communications between client and server must be reliable, efficient, scalable, and must be capable of handling network exceptions.

A user should be able to load the client component with a Web browser such as Netscape or Internet Explorer. The client component runs on the user's computer and needs to provide a complete graphical user interface (GUI) and keep some state information. The GUI should allow a user to enter expressions and parameters, to select from a number of options, and to view the resulting curves. The GUI must be straightforward and easy to handle. It should also provide input validation and should be able to inform user if any error occurs.

The server component runs on a host computer. It is called by the client and is responsible for parsing and evaluating expressions. The details of the expression formats and functional requirements of the parser will be described in Section 1.3. Once an expression is parsed or evaluated, the server component should return the results to the client. The server component must be efficient and reliable. In particular, the server must be able to report any singularities to the client.

Within each component, we need to follow the objected-oriented style. The object-oriented paradigm expounds three major ideas, namely, encapsulation, inheritance and polymorphism, which make system flexible, extensible, maintainable and easy to be reused.

Finally, users should be able to access the plotter from various platforms (Windows, Unix, Linux, etc.). The server is expected to be deployed on Windows/Windows NT, but it should be easy to port it to other platforms.

## 1.4 Detailed Requirement description

The following details the core functional requirements of the plotter.

- The plotter must be able to parse and validate an expression input by a user.

  An expression consists of identifiers (with one or more letters), constant numbers, function names (we support only built-in functions described in Section 1.2), and operators. Here are some typical expressions:

| | |
|---|---|
| *X^2+1* | ^ is the exponent operator |
| *(x-1)*(x+1)* | parentheses allowed; no implicit multiplication |
| *sin(x)+cos(x)* | function calls |
| *exp(x)^2* | $= (e^x)^2$ |
| *x - b* | blanks are allowed |

- The plotter must allow the user to define the role of identifiers:

  There must be exactly one independent variable; one identifier may be specified as a parameter; any other identifiers are assumed to be constants.

  The user must specify the range of the independent variable by entering a minimum and a maximum value. If there is a parameter, the user must specify the first value of the parameter, the final value, and the interval. Also, the user enters constants by specifying an identifier and a value. If an identifier is not defined by the user, by default, it is assumed to be a constant that has value zero.

- Once an expression is validated, and the identifiers are defined, the plotter should be able to evaluate the expression, and present the resulting curve or curves to the user. The plotter can be in one of the two states: the *initial* state and the *plotted* state. Before any curve is plotted, the plotter is in its *initial* state. When plotting the first graph, the plotter needs to find the minimum and maximum values in the desired range and choose suitable scales. After plotting a graph, the plotter is in *plotted* state. In *plotted* state, names (of variables, parameters, and constants) and scales on axes can no longer be changed. However, we can modify the values of a parameter or the value of a constant. In *plotted* state, the user may also introduce new expressions, and plot them in the same graph. In addition, the user can explicitly switch the plotter back to *initial* state by clearing the current graph.
- The program should detect obvious singularities such as:

  $a / b$ with b = 0

  *sqrt(x)* with x < 0

  *log(x)* with x<=0

  In these cases, the program should inform the user with a warning message and the corresponding points are not plotted.
- Other functionality:

  1. User is able to select the resolution of the graph (i.e., to choose how many points to plot).

  2. If the user clicks in the graph display area, the program responds by displaying the coordinates of that point.

  3. The user is able to choose colors for axes and the graph background.

The remainder of the report is organized as follows. In Chapter 2, the system design decisions will be addressed, including the architecture model, choosing appropriate technology and programming language, and the reuse of existing components. In Chapter 3, we will discuss user interface issues, such as tool selection and interface design. Chapter 4 summarizes the implementation highlights and how to deploy the application. Selected pieces of codes will be presented in the Appendix.

# Chapter 2

# System Design

## 2.1 Programming Language

A decision about the programming language to be used is not normally made at this stage of system design. For this project, however, we need to consider this issue early. The reason is that, many distributed technologies are language-dependent, and obviously the choice of technology has significant impacts on high-level system design, as well as other important issues such as budget and schedule.

As far as the requirements are concerned, Java programming language is the natural choice for implementing the client side of the application. Java is often referred to as the programming language for the Internet. It integrates perfectly well with both Web browsers and Web servers and thus provides excellent support for the development of Web-based applications. In particular, Java supports an applet model that allows the execution of Java code embedded in an HTML page. Specifically, Java applets are Java programs that are referenced from HTML pages and downloaded when a client requests the referring HTML page from the Web server. In the applet model, instead of invoking a method on a remote project, the code for the class providing the method is transferred across the network and executed locally by a *Java Virtual Machine* (JVM) inside the client's Web browser. In addition, Java applets provide excellent support for complex user interfaces through a variety of GUI class libraries such as AWT and JFC.

Based on the above features, we may implement Java applets as *thin clients*. A thin client provides a complete GUI, and keeps some state information (for example, in this project, it may cache current parameter settings, current plotting states, data of previous curves, etc.) as well as some application logic (for example, in this project, the rules of validating inputs, scaling, etc.). Since they contain state and some application logic, thin clients overcome the limitations of the so-called *null clients* (for instance, stateless HTML pages downloaded every time from a server) and send requests much less frequently to the server. Yet it needs to be stressed that Java thin clients are not expected to implement major application logic such as, in this example, expression parsing and evaluation. These are the responsibilities of the server component of the application.

One alternative (and perhaps, the only practical alternative for this project) to the Java-based design is to provide interactivity of traditional HTML-based interfaces through the Common Gateway Interface (CGI). CGI-based distributed applications, unfortunately, have a number of limitations and drawbacks:

- Clients (HTML) are stateless. The entire application (both server and the *real* client) actually executes on the server side. Consequently, there is no way to implement complex GUI on the client side and, to make things worse, the client side needs to communicate with the server side much more frequently than a thin client.
- Non-type-safe interaction. This is because HTML does not support typing, and all kinds of data are transferred as URL strings.
- Server has no clean mechanism to deliver notification to users.
- Performance bottlenecks. As a result of an invocation a complete HTML page is returned to the client side (including all the hidden state and GUI information). These

HTML documents contain a lot of repeated text and formatting data that remains unchanged since the last client action. The amount of unchanged HTML often outweighs the amount of actual data produced by the application program.

From these discussions, it is clear that Java is definitely a more flexible, clean and efficient solution for implementing the client-side component of the plotter application.

On the other hand, the programming language choice for the server side of the application is more debatable. Many programming languages can be applied to implement this component. Among them, C++ and Java are the most favorable choices for their supports of object-oriented programming, wide availability and popularity in industry. For this project, I've decided to use Java, for the following reasons:

- Java programs are highly portable due to the standardized byte-code representation generated by Java compilers. This makes Java the ideal language for component programming, since the deployment platform is not determined by hardware or operating system; the Java platform *is* the real deployment platform.
- Java provides a cleaner approach to object-oriented programming, with fewer memory management responsibilities, no pointers, a less confusing syntax, and simpler method resolution rules, etc.
- Most importantly, Java is supported by many distributed object technologies. As we need to rely on at least one of these technologies to implement this project, by using Java, we will have more alternative technologies to choose from. This issue will be further discussed in the next section.

## 2.2 Choosing Distributed Object Middleware

We can always develop a distributed client/server application directly using lower-level APIs, for instance, sockets. However, this approach is not only tedious but also error-prone. For a rapid and effective development of distributed application, we need the aid from *middleware*, a kind of software that provides a framework to shield programmers from lower level details of networking protocols, as well as heterogeneous data representation, hardware, and software environments across networks. A middleware also supports a number of features and services to help managing distributed environments.

For this project, we need to select a proper middleware based on distributed object technology. *Distributed Objects* refers to a means of building distributed applications based on object technology. Through the characteristics such as abstraction, encapsulation, inheritance and polymorphism, a distributed object model provides an effective way to manage and reduce system complexities, reduce cost, and increase flexibility and extensibility. A distributed object middleware provides a distributed object infrastructure so that we can build and deploy systems in which objects may be used independent of the platform, the network transport, the object implementation details and the locations of objects in the network.

### 2.2.1 DCOM, CORBA and RMI

The next step is to choose a distributed object middleware from the three most popular and powerful alternatives: Distributed Component Object Model (DCOM) [6], Common Object Request Broker Architecture (CORBA) [7,8] and Java Remote Method Invocation (RMI) [9]. An in-depth layer by layer comparison between DCOM and CORBA can be found in [10]. Raj, on the other hand, has compared DCOM, CORBA and RMI from a programmer's perspective [11].

DCOM is the distributed extension to COM (Component Object Model) developed by Microsoft. With DCOM, the client first acquires a pointer to one of the server's interfaces. Through the acquired interface pointer, the DCOM client calls into the exposed methods of the server object as if the server object resided in the client's address space. Remote access in DCOM relies on a protocol called the *Object Remote Procedure Call* (ORPC). The ORPC layer is built on top of DCE's RPC and interacts with COM's run-time services.

DCOM offers a number of distinct benefits. For example, it supports multiple interfaces per object; it supports binary software integration and large-scale, cross-language software reuse. The main problem with DCOM is that it is tightly integrated with the Windows platform, and heavily relies on its registry system, COM services and security model. Since we don't want limit this application to Microsoft platforms, the use of DCOM is excluded.

CORBA is a standard of distributed object model proposed by the largest software consortium in the world -- the Object Management Group with over 1000 members. CORBA allows objects on one machine to communicate with objects running on any number of different machines via the *Object Request Broker* (ORB). At a lower level, it depends on a protocol called the *Internet Inter-ORB protocol* (IIOP) for remote accessing.

The most significant advantage of CORBA is that it is a very open standard, which is neither operating-system specific nor programming language dependent. The CORBA objects can be written in virtually any language and can run on virtually any platform. CORBA is currently the only technology that supports such a broad range of languages and hardware environments. Besides, CORBA also has many other benefits. For example, CORBA provides a wide range useful services such as naming services, event services and lifecycle services. As well, CORBA is scalable for large systems.

However, CORBA also has some disadvantages. Above all, CORBA is only a standard specification. To build our application on CORBA, we first need to purchase a specific CORBA ORB product (for example, VisiBroker from Borland or Orbix from Iona Technologies) and install it. The ORB products are typically very expensive and may use a lot of computer resources. This is fine if we are building a large, complex enterprise application. For this small application of 2-D plotter, however, the requirements simply don't justify the cost. Second, on the client side, user needs to download the CORBA ORB at run-time. This may significantly affect the performance of the application. Although some browsers already include an ORB of some kind (for example, VisiBroker 2.5 is shipped with Netscape 4.x), others don't. Even for those browsers that do include ORB, they are only compatible with a particular ORB product and they may not support its latest versions. Third, despite the fact that OMG is aiming at solving the problem, the interoperability of different ORB products continues to be a big issue. (For example, a VisiBroker client may not communicate properly with an Orbix server). To make things worse, ORB vendors update their products frequently while their support for back compatibility are poor. Consequently, an application built on CORBA will very likely be bound to an ORB product from a particular vendor and even to a particular version of that product.

To conclude, although technically, CORBA can provide a solution for this application, choosing CORBA as our middleware is against our goal of building a cheap, simple, efficient and flexible plotter.

RMI is JavaSoft's solution for distributed object computing. RMI seamlessly supports remote method invocation on objects across Java Virtual Machines. It relies on Java Object Serialization for marshaling object data and a protocol called the *Java Remote Method Protocol* (JRMP) for remote access. More technical details of Java RMI will be given in the next section.

RMI is a pure Java solution and is specifically designed to operate in the Java environment. With RMI, all client and server objects must be written in Java. This makes it more difficult to reuse a legacy system written in other programming languages. This problem, however, can be solved by a Java technology called *Java Native Interface* (JNI). We will discuss this issue in Section 2.3.

Another problem with RMI is that the performance of a RMI-based application might become slow when it is scaled. This, however, is not a severe problem for this particular application. While the plotter should be able to perform well in a multiple-user environment, it is not expected to be accessed by thousands of users concurrently (it is, after all, not a database application).

Despite the above limitations, RMI exhibits many distinct advantages:

- RMI is smoothly integrated into Java and Web environments.
- Other technologies, such as DCOM and CORBA, require mappings from an interface description language (IDL) to a particular programming language. This is not needed

for RMI. In RMI, component interfaces are defined in Java directly. It is thus type-safe and easy to understand.

- It is simple and easy to program.
- It can be used on diverse operating systems, as long as the platform has Java support. It allows cross-platform execution in browsers.
- It supports passing object *by value*, which allows moving the code of the class from one JVM to another. As a result, accessing the object will be always be a local operation, which reduces the network workload. This feature is particularly useful for the plotter application, since the server can wrap all the results (all data, ranges, data size, error information, singularity information, etc.) into a single object and return it to the client in one shot. (In contrast, CORBA normally passes object *by reference*. Although latest CORBA specification has introduced *pass by value*, it is not yet well supported by vendors.)
- Last, but probably most important, RMI is included in JDK. Since JDK can be downloaded for free, using RMI means no additional cost.

From the above analysis, it is clear that Java RMI is the most suitable technology to develop this project. Hence, RMI is chosen as the middleware for our distributed 2-D plotter application.

### 2.2.2 RMI Overview

RMI is an extension of Remote Procedure Call (RPC). RPC abstracts the communication interface to the level of a procedure call. RPC, however, does not translate well into distributed object systems, where communication between program-level objects residing in different address spaces is needed. On the other hand, RMI directly integrates a distributed object model into the Java language. To accomplish this, RMI introduces local proxy objects to manage the invocation on a remote object.

In the terminology of RMI, the proxy object which resides on the client side is called a *stub*. The proxy on the server side is called a *skeleton*. The client can easily communicate with the stub because they are in the same Java virtual machine. Likewise, the remote server can easily communicate with the skeleton. The stub acts as a surrogate for the server so that the client can access the server transparently. When the stub receives a message from the client, it forwards the request and marshals the parameters across the network to the server, obtains the server's response, and returns it the client. The skeleton performs a similar function on behalf of the server. The communication between stub and skeleton is based on the protocol of JRMP. At a lower level (the transport layer), JRMP relies on TCP protocol by default. However, by installing a custom RMI socket factory, it is possible to let the RMI transport layer use a non-TCP or custom transport protocol over IP. Figure 1 sketches how the plotter client and plotter server components communicate via stub and skeleton.

The codes for stub and skeleton can be automatically generated. To do so, first we need to define the remote interface for a server object. The interface exposes a set of methods that are indicative of the services offered by the server object. Once the interface is defined, we can use a tool provided by Java to generate the proxy objects.

Before the client can invoke a method of the remote server object, it must obtain its object reference. RMI provides a *registry* service that runs on the server machine. The registry maintains a database of named remote objects. When a client wants to use a remote object, it asks by name for the reference to the remote object. The registry scans its database and returns a reference to the requested object. Figure 1 also shows how RMI's registry works. Once the client obtains the object reference, it can call a method implemented by the server object as if the server were running on the same Java virtual machine.

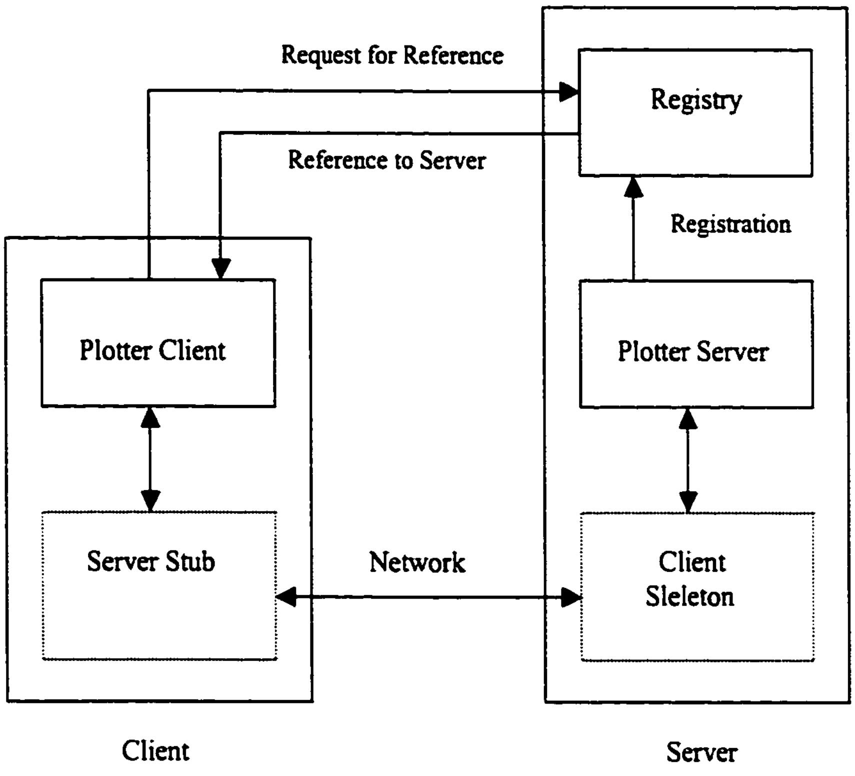


Figure 1: The plotter client obtains reference to a server object via RMI registry and communicates with the plotter server via stub and skeleton .

### 2.3 Parser

For this project, it was decided to reuse the parser module that had been used in Tran's project. As mentioned in Section 1.2, the parser was built using the *recursive descent method* [5]. It performs the following tasks:

- Given an expression, determine if it is syntactically correct.
- Given the values of the independent variable, the parameter and the constants in the expression, evaluate its value.

Details of the parser can be found in reference [4].

The main difficulty of reusing this parser lies in the fact that the parser was written in C++, while the current project is intended to be written in Java. Fortunately, Java provides the Java Native Interface (JNI) facility [12] that allows us to build a Java interface to a non-Java legacy system. Specifically, the JNI is a native programming interface. It is organized like a C++ virtual function table, through which Java code running inside a Java virtual machine can interoperate with applications and libraries written in other programming languages, such as C, C++ and assembly. With JNI, the non-Java legacy system (in this case, the parser module) can be integrated in the form of a dynamic link library (DLL).

From the standpoint of software engineering, the reuse of the parser component reveals many benefits. First, it eases the development efforts significantly. To reuse this parser, we only need to understand its interface. The internal details of the parser are no longer our concern. Second, since the parser module has already been tested extensively, the reuse of the parser makes the current application more reliable. Third, as the parser and the rest of the system are clearly separated, they can be maintained and if necessary, extended independently. Fourth, it is well known that Java is more powerful for Internet programming, while C++ is faster, more efficient, and more flexible in performing conventional but complex tasks such as parsing and mathematical computation on a local machine. In this project, by reusing the parser written in C++, we are combing and taking advantages of the strong points of both programming languages.

## 2.4 System Architecture

Based on the discussions in 2.1 - 2.4, Figure 2 summarizes the overall architecture of the plotter application. The functionality and the details of each component are described as follows.

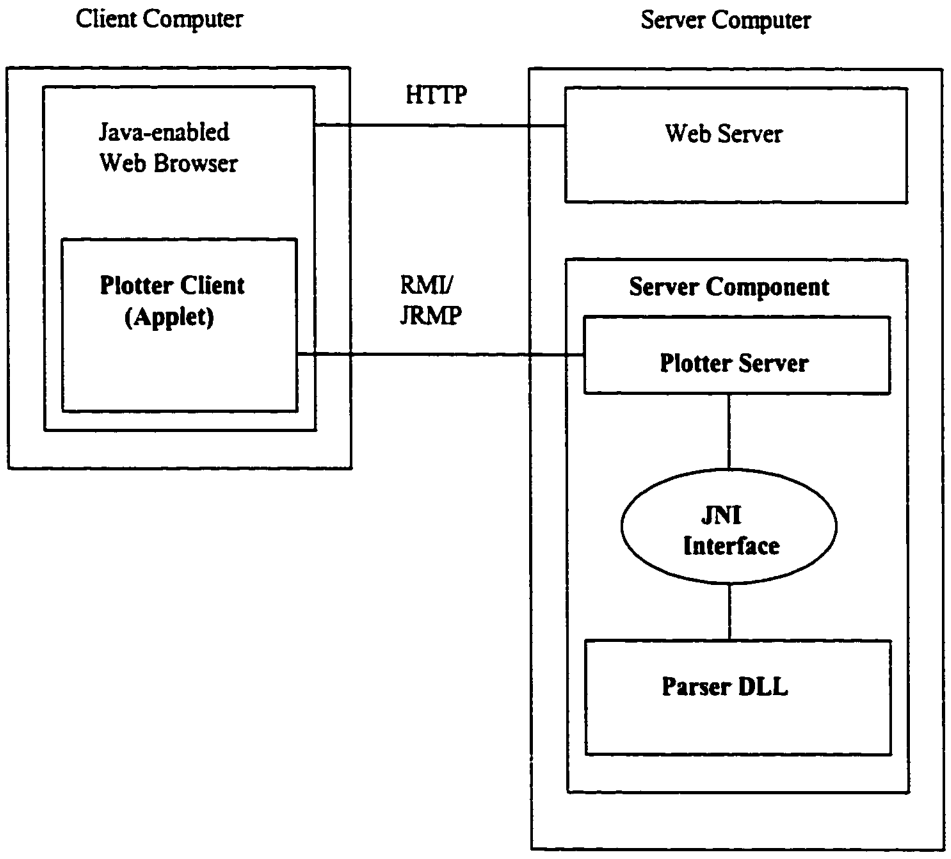


Figure 2: The major components of the plotter application.

- *PlotterClient*: PlotterClient is the client-side component of the application. It consists of a *RectangleArea* class that presents a *view* on which curves can be plotted, a number of auxiliary classes, each of which represents an element in the graphical user interface, and a *PlotterClient* class that manages the above classes. The *PlotterClient* needs to be executed inside a Web browser, and communicates with *PlotterServer* via RMI.

- *PlotterServer*: *PlotterServer* component is the server-side component of the application. It accepts the requests from the *PlotterClient*, uses its application logic to processes the requests and returns the results to the client. When processing the requests, *PlotterServer* needs to call the *Parser DLL* to parse or evaluate an expression. *PlotterServer* consists of three parts. The *PlotterServer* class is the core of the component; the *PlotterServerInterafаce* defines the RMI interface; and the *Set* class records the current request from a client.
- *Parser DLL*: This is the reused parser. Its responsibilities have been described in Section 2.4.
- *JNI interface*: The JNI interface declares the native methods that allow the *PlotterServer* to access the Parser.

In addition, we need to implement the following elements:

- *Plotter.html*: An HTML page into which the *PlotterClient* applet is embedded.
- *Results*: An instance of this class wraps all the information that is returned to the *PlotterClient* by the *PlotterServer*. The information includes the coordinates of all the points on the curves, the ranges of the coordinates, the error messages (parsing error or singularity information, if any). With RMI, a *Results* object can be parsed to the client *by value*, in one shot.

# Chapter 3

# User Interface

## 3.1 JFC (Swing) vs. AWT

Providing a friendly graphical user interface (GUI) is one of the major responsibilities of the client component. For this application, the goal of the user interface is simple, easy to learn and easy to use.

Java provides a rich set of classes for GUI development. These classes are grouped in the *Abstract Window Toolkit* (AWT). AWT is a powerful tool for developing GUI, but it uses the facilities of the host platform to create and display user interface controls. On one hand, this allows the AWT preserves the familiar look and feel of the host platform. On the other hand, this makes it difficult to write portable program using the AWT, owing to subtle differences from platform to platform in the behavior of native controls. Furthermore, AWT components are restricted to the least common denominator of the features that are present on every platform.

To overcome limitations of the AWT, *Sun* created the *Java Foundation Classes* (JFC, also known as *Swing*) [13]. The biggest difference between the AWT components and Swing components is that the Swing components are implemented with absolutely no native code. As a result, a GUI developed by Swing may maintain a consistent look and feel across platforms. Moreover, it lets programmer specify which look and feel a program's GUI uses. Because it does not use native code, the JFC completely controls drawing all its components. The Swing components therefore are not restricted to the features supported by the native platform, and as a result, have capabilities far beyond what the AWT components offer. For example, Swing buttons and labels can display images; the borders around most Swing components can be easily added or changed; the behavior or appearance of a Swing component can be easily changed; Swing components don't have to be rectangular. Although for now we may not need these features in this application, it is good to leave the possibilities open for future extension. A feature of JFC that we do need for this project is that, with JFC, we can use the "menu" system in Java applets. (In contrast, with AWT, we can only use menu in standard Java application.) By introducing menus, we may develop a simple, straightforward, yet powerful GUI. Without menus, we would be forced to place many textboxes and buttons on the front page, which makes the user interface very clumsy.

Based on the above benefits, JFC (Swing) is used to develop GUI for this application.

## 3.2 User Interface Description

### 3.2.1 The Main Interface

Figure 3 shows the main user interface of the plotter. As can be seen, it is embedded in a Web browser. It consists of three parts: a menu bar on the top, a rectangular display region in the middle, and an information bar at the bottom.

The *menu bar* allows user to enter expression, set variable range, parameter range, and constants, retrieve current settings, and take actions such as plotting, clearing, etc. We will describe it in detail in 3.2.2.

The *information bar* displays the coordinates of the point if the user clicks in the graph display area. This helps user to understand the expressions being drawn, and obtain the data along a curve or at the intersections of multiple curves.

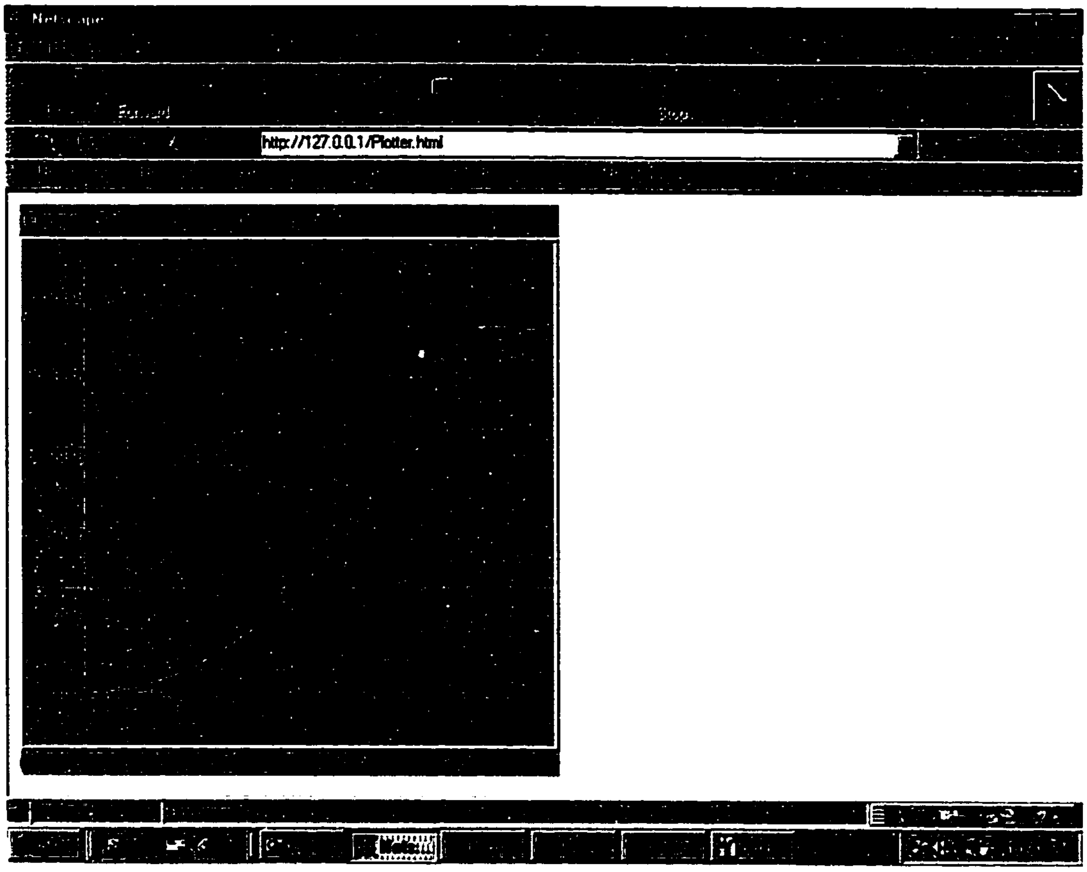


Figure 3: The main user interface of the plotter

The display area is used to draw the curves based on the expressions specified by user. When the applet is started, the *display area* is nearly blank with only bare X and Y axes on it. When the first curve is drawn, labels and scales are written on the axes, based on the data ranges of the curve and the name of the independent variable. Multiple curves (either for different expressions, or for the same expression but different parameter or constant values) can be drawn on the same view, each with a different color. User can clear the display area at any time by selecting a proper menu item.

### 3.2.2 The Menu

The menu bar contains four menus: *Curves*, *Settings*, *Option* and *Help*.

#### 3.2.2.1 Curves

Clicking on *Curves* with the left mouse button displays a pull down menu, as shown in Figure 4. It provides two menu items: *New* and *Plot*. Selecting *Plot* draws a new curve. If no valid expression is currently specified or the independent variable is not set, an error message will be displayed in a separate window. In addition, if any singularities are detected, a warning message window will be popped up (while the rest of the curve is still drawn). Clicking *Plot* also moves the program from the *initial* state to the *plotted* state (see the section 1.3 for the definitions and descriptions of these two states).

Clicking *New* does the opposite. It clears the display area, and if the program is currently in the *plotted* state, it resets it to the *initial* state.

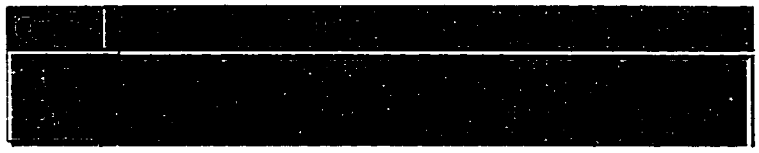


Figure 4: The *Curves* Menu

#### 3.2.2.2 Settings

This menu allows user to enter an expression, to define variable, parameter, or constants, and to specify their values or ranges of values. As shown in Figure 5, It has six menu items: *expression*, *variable*, *parameter*, *constants*, *undefine*, and *current*.

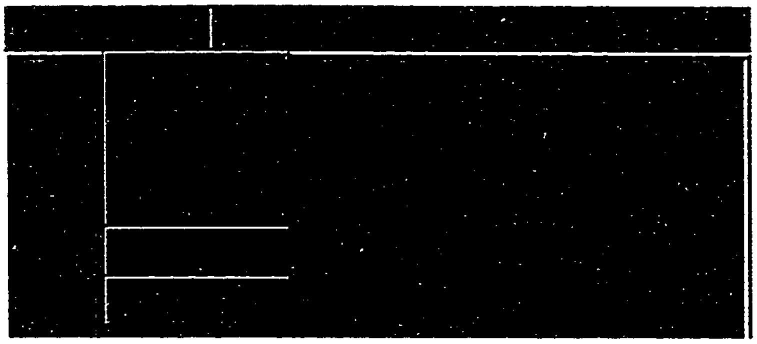


Figure 5: The Settings Menu

*Expression*: Selecting this menu item brings up the dialog box shown in Figure 6. It allows user to enter an expression. By clicking the *Parse* button, the expression will be sent to the server and parsed there. After the expression has been parsed, a message window will be displayed to let user know whether the expression is valid, as shown in Figure 7.

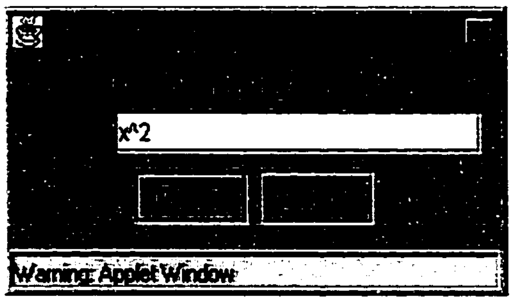


Figure 6: The dialog box for entering expression

(Note: the warning message is a Java security feature [14] that prevents Applet windows from initiating other applications.)

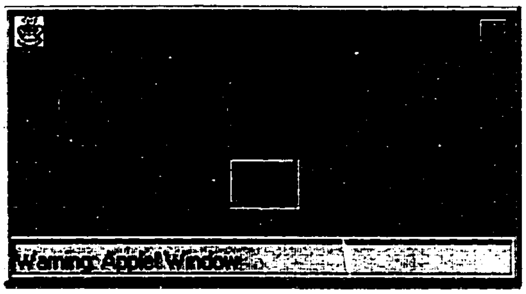


Figure 7: The message window showing the parsing result.

*Variable:* Selecting the *Variable* menu item brings up the dialog box shown in Figure 8. It allows the user to specify the name and range of the independent value. The input is validated. For example, if the name does not exist in the expression, an error message will be displayed.

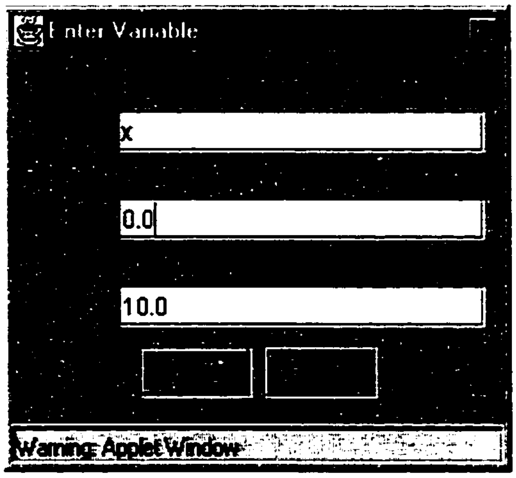


Figure 8: The dialog box for defining variable

*Parameter:* Selecting the *Parameter* menu item brings up a "Enter Parameter" dialog similar to the one shown in Figure 8. The only difference is that it has one more text box to let user specify the step for the parameter.

*Constants:* Selecting the *Constants* menu item brings up a "Add Constant" dialog box that allows user to add one constant at a time, by specifying its name and value.

*Undefine*: Sometimes user may wish to change the definition of an identifier, for instance, from "parameter" to "constant". To do so, user must undefine the identifier first and then define it again. Selecting the *Undefine* menu item brings up a dialog box that allows user to undefine an identifier by specifying its name, as shown in Figure 9.

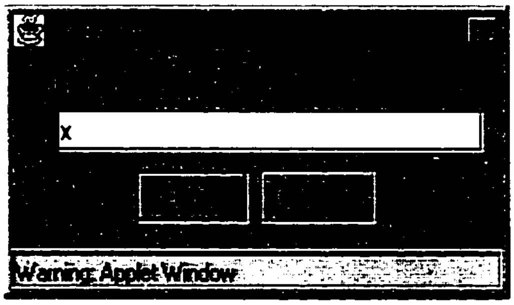
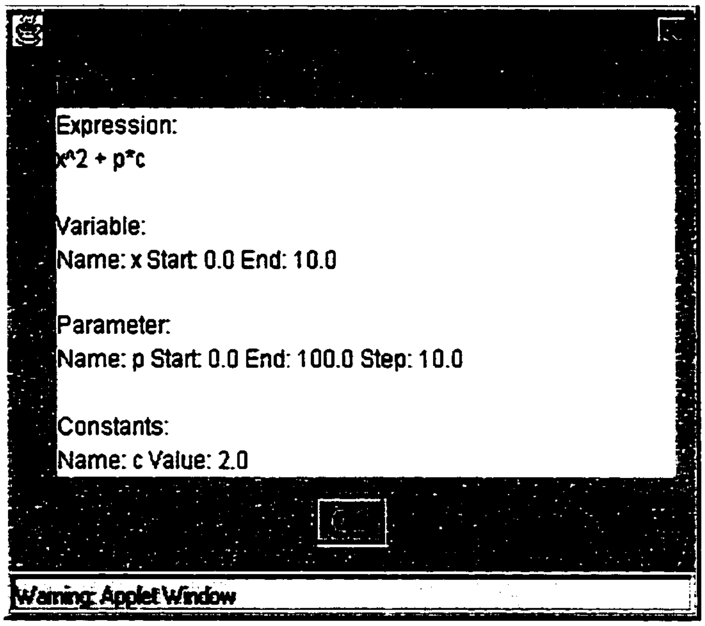


Figure 9: The *Undefine* Dialog Box.

*Current*: Selecting the *Current* menu item brings up a window showing the current settings, including the expression, the name of the variable and its range, the name of the parameter, its range and its step, and the names and values of the constants. (Figure 10)



Expression:
x^2 + p*c

Variable:
Name: x Start 0.0 End: 10.0

Parameter:
Name: p Start 0.0 End: 100.0 Step: 10.0

Constants:
Name: c Value: 2.0

Figure 10: The window showing the current settings.

### 3.2.2.3. Options

The *Options* menu has two menu items: *Color* and *Resolution*, as shown in Figure 11.

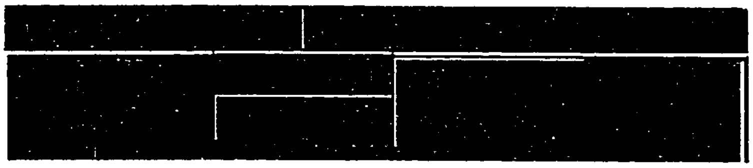


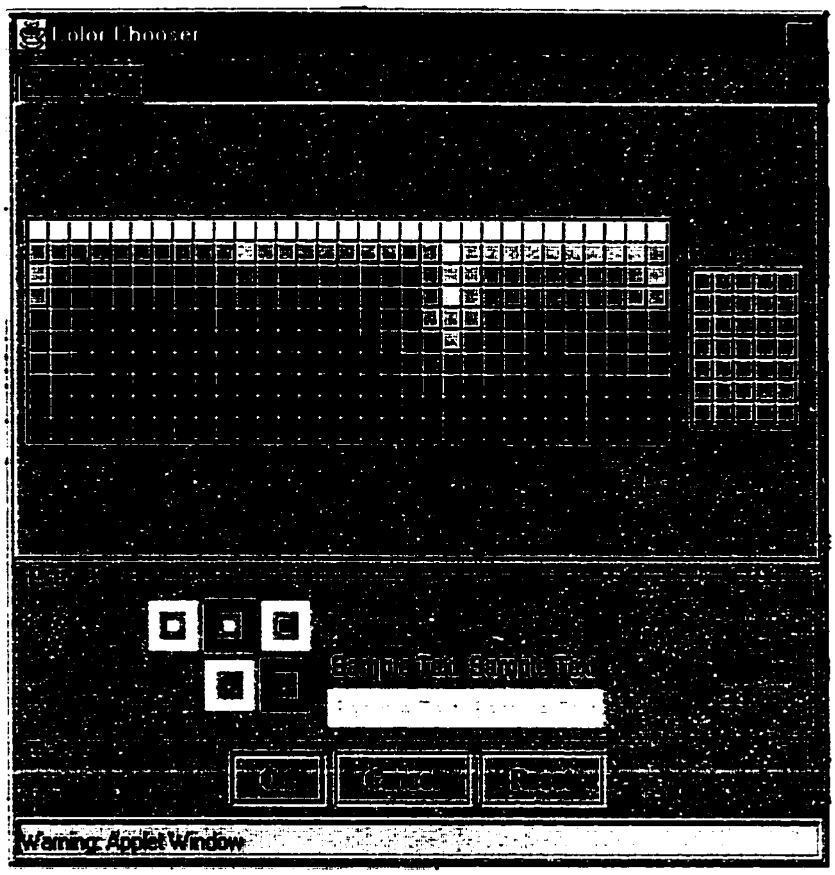
Figure 11: The *Options* Menu

<u>*Color*</u>: The menu item has two submenus: *Background* and *Axes*, which allow user to change the background color of the display area and the color of the axes in the display area respectively. Selecting either of them brings up a "color chooser", as shown in Figure 12. Clicking on a color and pressing "OK" will change the corresponding color in the display area.

<u>*Resolution*</u>: Selecting this menu item brings up a dialog box that contains a radio button group to let user specify the plotting resolution, as shown in Figure 13. With higher resolution, the curve is smoother but the drawing is slower. With lower resolution, the curve is rougher but it is faster. By default, the resolution is set at "Medium", which draws a point for every five pixels. For the "High" resolution, a point is drawn for every pixel. For the "Low" resolution, a point is drawn for every twenty pixels.

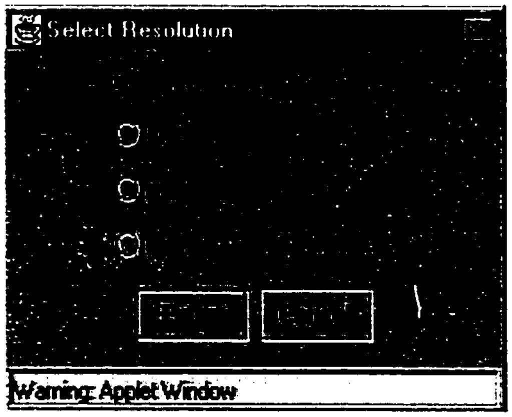
### 3.2.2.4 Help

The *Help* menu has only one menu item: *Summary*. Selecting this menu item brings up a window that summarizes the functions of all the menu items described above. It is intended to help users to learn this tool quickly.



Warning: Applet Window

Figure 12: The color chooser



Warning: Applet Window

Figure 13: The select resolution dialog box

# Chapter 4

# Implementation and Deployment

## 4.1 Defining the Server Interface

The first step of developing a client/server application with RMI is to define the remote interface. The interface declares the functions that can be invoked remotely by a RMI client. The following shows the *PlotterServerInterface* written in the Java language.

```
import java.rmi.*;
public interface PlotterServerInterface extends Remote
{
      public boolean checkExpr(String xpr) throws RemoteException;
      public boolean setParameter(String name, double first,
                  double last, double step) throws RemoteException;
      public boolean setVariable(String name, double min, double max)
                  throws RemoteException;
      public boolean setConstant(String name, double val)
                  throws RemoteException;
      public Results evaluate(double step)
                  throws RemoteException;
      public void clear() throws RemoteException;
}
```

As can be seen, the remote interface defines the signatures of six functions:

*checkExpr*: allows the client to specify the expression. The server, in turn, parsers the expression, and returns a *boolean* to indicate whether the expression is acceptable by the parser's grammar.

*setParameter*: let the client set the parameter for the expression, by specifying its name, start value, end value, and step. The function returns a *boolean* value to indicate whether the name is valid. (It returns *false* if the name is not in the current expression.)

*setVariable*: let the client set the independent variable for the expression, by specifying its name, start value, and end value. The function returns a *boolean* value to indicate whether the name is valid, as an element of the current expression.

*setConstant*: let the client set a constant for the expression, by specifying its name and its value. The function returns a *boolean* value to indicate whether the name is valid.

*evaluate*: method evaluates the current expression with the current settings of variable, parameter and constants, and returns the results to the client. The argument "step" corresponds to the resolution of the evaluation.

*clear*: method cleans up the current settings on the server side. It makes the server ready to accept a new expression.

Here several things are worth noticing. First, the remote interface extends the *java.rmi.Remote* interface, as is required by RMI. Second, each method declares *java.rmi.RemoteException* in its *throws* clause. Because remote method invocations can fail in very different ways from local method invocations, remote methods will report communication failures by throwing a *java.rmi.RemoteException*. Third, the *evaluate* method returns a *Results* object, rather than a simple array of data. The *Results* encapsulates the resulting data of the evaluation and several methods to help the client to retrieve the data. The data include the coordinates of all the points, number of curve segments (since the expression can have a parameter, the result may contain several curve segments), the total number of points, error messages (including singularity information), etc. The methods, for example, help client to find the maximum and minimum values in the X and Y directions. RMI supports *passing by value*. Therefore, the methods in the *Results* can be accessed locally by the client. To allow so, however, the *Results* class must implement the *Serializable* interface. (In Java, the *Serializable* interface is a marker interface. Objects implementing the *Serializable* interface can be written to streams.)

## 4.2 Client Implementation

*PlotterClient* is implemented as a Java applet. It manages three GUI components: a menu bar (*mb*), a label (*label*) and a display area (*rectangleArea*). The label is a *JLabel* object, and is used to show the coordinates of a point in the display area when user clicks it. The menu bar is implemented by a *JMenubar* object, which contains a set of *JMenu* objects. Each *JMenu* object handles a menu. A *JMenu* object is further associated with one or more *JMenuItem* objects. The *PlotterClient* implements Java's *ActionListener* interface, and whenever user selects a menu item, it invokes a proper event handler. There are two major categories of tasks that might be performed by an event handler. One type of task is to invoke a method of the *rectangleArea* object, which, in turn, plots curves on the display area, or communicates with the server. The other type of task is to pop up a dialog box to let user enter inputs. A dialog box is represented by an instance of a class that inherits the *JDialog* class. For example, for variable input, we have the *VariableInput* class, for parameter input, we have the *ParameterInput* class, and for selecting resolution, we have the *ResoInput* class, etc. Each of these classes contains a set of GUI elements, such as text fields (*JTextField*), buttons (*JButton*), radio buttons (*JRadioButton*). These classes are also responsible for validating user inputs, and furthermore, each of them stores a reference to the owner *PlotterClient* object, via which it may send validated user inputs back to the *PlotterClient*.

A *RectangleArea* class inherits the *JFC* class *JPanel*. It has two responsibilities:

- It manages the communication with the server.
- It defines a display area and draws the graph on it.

To communicate with the server, in its constructor, the *RectangleArea* obtains a reference to its remote server from the server host's RMI registry (known as *rmiregistry*), as follows:

```
class RectangleArea extends JPanel {
      PlotterServerInterface server;
      static final String URL = "rmi://localhost/plotter";
      -- --
      public RectangleArea(PlotterClient controller) {
            this.controller = controller;
            try{
                  server =
                     (PlotterServerInterface)Naming.lookup(URL);
            }
            catch (Exception ex) { fatalError(ex); }
            -- --
      }
      -- --
}
```

The static *lookup* method of the *Naming* class (imported from the import *java.rmi.Naming* package) takes an URL-formatted name, and returns a reference to an object that implements the *Remote* interface. The constructor then casts the returned value to a *PlotterServerInterface*.

Once the *RectangleArea* obejct obtains a reference to the server, it may act as a bridge between the user interface (*PlotterClient*) and the server (*PlotterServer*). It has a number of functions such as *parseExpr*, *setParameter*, *setVariable*, *setConstant*, and *clear*. Whenever the *PlotterClient* receives user inputs, it invokes one of these methods, and the invoked method, in turn, calls the corresponding method declared in the remote interface *PlotterServerInterface*. Note that the *RectangleArea* also stores a reference (called *controller*) to the *PlotterClient* object so that it can return the *boolean* flag (successful or not) returned by the server to the *PlotterClient*. The *PlotterClient* is then responsible for displaying this result to the user.

The *RectangleArea* implements a *plotCurve* method. Whenever user selects the *Plot* menu item, this method is invoked by the *PlotterClient*. The *plotCurve* then calls the server's *evaluate* method. As mentioned in 4.1, the *evaluate* method takes a parameter representing the computing step. Thus, before calling this method, the *RectangleArea* needs to compute this argument from the resolution (in pixels) specified by the user and the scale in the X-direction (how much a pixel represents). The *evaluate* method returns a *Results* object. In order to draw multiple curves in a single view, the *RectangleArea* stores an array of *Results* objects. Whenever a new *Results* object is returned, it is added to the array. Then the *plotCurve* method calls the *repaint* method, which, in turn, implicitly invokes the *paintComponent* method to update the display area.

The rectangle also implements a set of auxiliary methods to assist graph plotting, listed as follows.

_drawAxis_: this function draws the X and Y axes, and writes axis names. It also draws the ticks for the axes, and writes the labels for the ticks.

_formatText_: this function is used to format tick labels. For very small or large numbers, we use scientific notation. Also, for the sake of clarity, a number might be rounded so that no more than two significant digits are displayed.

_computeScale_: this function finds suitable drawing scales from the minimum and maximum values in the desired range.

Besides, *RectangleArea* implements a *MouseListener* that can catch any mouse-clicking event in the display area. The *MouseListener* then calls the *showCoordinate* function to display the coordinates of the point that is clicked.

### 4.3 Server Implementation

On the server side, a *PlotterServer* object is responsible for communicating with client, implementing the application logic, and connecting to the parser component.

#### 4.3.1 Communicating with the Client

In order for the PlotterServer to communicate with the PlotterClient, it must register itself in the RMI registry, as shown by the following source code.

```
public class PlotterServer extends UnicastRemoteObject
      implements PlotterServerInterface {
      static final String SERVERNAME = "//localhost/plotter";
      ... ...
      public static void main(String [] args) {
```

```
                System.setSecurityManager(new RMISecurityManager());
                Try {
                      PlotterServer server = new PlotterServer();
                      Naming.rebind(SERVERNAME, server);
                      System.out.println("Server ready.");
                }
                catch (Exception ex){
                      ex.printStackTrace();
                }
                -- --
          }
          -- --
    }
```

As is shown, the *main* method first creates an instance of the *PlotterServer* object, then binds a URL-formatted name of the form "*//host/objectname*" to the object using *Naming* class's static method "*rebinding*".

Also shown in the above code, the server needs to create and install a security manager. For simplicity, here the *RMISecurityManager* is installed. However, it is also possible to install a user-defined manager. A security manager needs to be running so that it can guarantee that the classes that get loaded do not perform operations that they are not allowed to perform.

Finally, the *PlotterServer* class extends *java.rmi.server.UnicastRemoteObject*. This makes an instance of the server class be exported automatically upon creation. Once it is exported, the object is ready for accepting incoming remote method calls.

### 4.3.2 Reusing the Parser Component

The reused parser component offers the services of parsing and evaluating expressions through the following methods of its *ParsEval* class:

*void parse (char *expression, bool & ok)*: it attempts to parse the given string, and set 'ok' to 'true' if successful, 'false' otherwise.

*bool set (char *name, double value)* : it sets the value of the variable 'name' to 'value'.

*double eval ()*: it evaluates the expression using current values of variables.

As mentioned, the parser was written in C++. In order to use them from the Java class *PlotterServer*, we need to take advantage of the *Java Native Interface* (JNI) technology. With JNI, first, we need to define a Java class *ParseInterf*. This class contains a set of native methods that can be called by the PlotterServer class, including:

- public native int ParseExpr(String jexpr);
- public native int SetVariable(String jname, double value);
- public native double EvalExpr() throws IllegalArgumentException;

It also loads the *parser* library with the *System.loadLibrary* method. On the C++ side, a file named *ParsInterfImp.cpp* is written. Its implements the following methods in C++:

*JNIEXPORT jint JNICALL Java_ParsInterf_ParseExpr (JNIEnv *env, jobject obj, jstring jexpr)*: this function calls *ParsEval*'s *parse* method, and can be called by the *ParseExpr* method of the *ParseInterf* class.

*JNIEXPORT jint JNICALL Java_ParsInterf_SetVariable (JNIEnv *env, jobject obj, jstring jname, jdouble value)*: this function is a bridge between the *set* method of the *ParsEval* class and the *SetVariable* method of the *ParseInterf* class.

*JNIEXPORT jdouble JNICALL Java_ParsInterf_EvalExpr(JNIEnv *env, jobject obj)*: this function is a bridge between the *eval* method of the *ParsEval* class and the *EvalExpr* method of the *ParseInterf* class.

As indicated from above, in *ParsInterfImp.cpp*, Java data types are mapped to their machine-independent native equivalents, for example, Java's *int* is mapped to *jint*, *String* is mapped to *jstring*, *double* is mapped to *jdouble*.

To connect the *ParseInterf* class and *ParsInterfImp.cpp*, we need to have a head file called *ParsInterf.h*. This file can be generated by the javah tool of JDK with the following two steps:

- Compile *ParseInterf*: `javac ParsInterf.java`
- Create *ParsInterf.h* from the *ParseInterf* class: `javah -jni ParsInterf`

Finally, the *ParsInterfImp.cpp* and the parser module (*parseeval.cpp*) are built together into a library (called *parser*). For example, in Windows platform, they can be built into a *DLL* with the Visual C++ command line compiler, as follows:

```
prompt> cl  -Ic:\jdk1.2.2\include  -Ic:\jdk1.2.2\include\win32  -LD  -GX
parseval.cpp ParsInterfImp.cpp -Feparser.dll
```

### 4.3.3 Implementing the Application Logic

The *PlotterServer* class implements the remote interface *PlotterServerInterface*. In other words, it provides methods bodies, or definitions, for each of the method signatures declared in that interface.

The method bodies implement the application logic of the plotter. This can be demonstrated by the implementations of the *evaluate* method. As shown in 4.3.2, the parser treats all identifiers equally as variables. As a result, it is the *evaluate* method that is responsible for distinguishing the constants, the parameter, and the real independent variable. It uses the following logic to ensure all the points across the entire ranges of the variable and the parameter are evaluated correctly. Suppose that there are *n* constants (the $i^{th}$ constant has name *ci*, and value *vi*), and there is one parameter *p* (starting at *ps*, ending at *pe*, with a step of *pt*). Also suppose that the independent variable is named as *v*, which starts at *vs*, ends at *ve*, and with a step of *vt*. The *evaluate* method implements the following loops:

```
Initialize a new Results object.
for (int i=0; i<n; i++)
      Call ParsInterf's SetVariable with (ci, vi);
int j = 0;
While (ps + j * pt < pe)
      Call ParsInterf's SetVariable with (p, ps + j * pt);
      int k = 0;
      While (vs + k * vt < ve)
            Call ParsInterf's SetVariable with (v, vs + k * vt);
            Call ParsInterf's evalExpr();
            Add the value returned by evalExpr() to the Results;
            k++;
      j++;
```

After all the loops are executed, the *evaluate* method returns the *Results* object. The *evaluate* method also handles singularities. When the parser encounters a singularity point, it raises an *Error* exception that contains an error message. The exception is caught by the *Java_ParsInterf_EvalExpr* method defined in the *ParsInterfImp.cpp*. This method, in turn, throws a *java.lang.IllegalArgumentException* that contains the original error message. Finally, this exception is caught by the *evaluate* method. The *evaluate* method then extracts the error message and inserts it into the *Results* object. It also sets a flag in the *Results* object to indicate that singularities occur. When the *Results* object is returned to the client, the client displays the error message to user.

## 4.4 Deployment

How to deploy an application is an important issue in the distributed environment. This section details the steps to deploy the plotter project. It is assumed that the task is performed on Windows/NT platform and the web server involved is Microsoft's IIS or PWS (Personal Web Server). However, for other platforms, or web servers, the steps are similar.

### 4.4.1 Prepare HTML file

To allow user load the *PlotterClient* applet in a Web browser, the applet must be embedded in an HTML file (*Plotter.html*) as follows.

```
<APPLET  CODE = "PlotterClient.class" CODEBASE = "myClasses/"
WIDTH = 400 HEIGHT = 400 ></APPLET>
```

The *CODE* specifies the name of the applet. The *CODEBASE* specifies the directory below the directory from which the web page was itself loaded. All the Java class files to be loaded should reside in this directory. *WIDTH* and *HEIGHT* specify the size of the applet in the browser, in pixels.

In order to use Swing applet (and in fact, many other features and capabilities of JDK, such as RMI, JNI, JaveBeans), *Java Plug-in* must be installed on the browser. The Java Plug-in is a software product from Sun Microsystems Inc. It enables web page authors to direct Java applets on their web pages to run using Sun's *Java Runtime Environment*

(JRE), instead of the browser's default Java runtime. The Java Plug-in can be downloaded for free [15].

To use Java Plug-in to run applets on web pages, applet tags on the HTML pages must be converted to the format required by Java Plug-in. This can be done either manually or automatically by the HTML Convertor tool. The tool is also made available as a free download by Sun Microsystems Inc.[15] .

The converted *Plotter.html* is listed in the APPENDIX of this report. This file needs to reside in the web server's public HTML directory (for IIS, or PWS, this directory, by default, is *inetpub\wwwroot.*).

### 4.4.2 Prepare Security Policy File

To enforce RMI security, a policy file needs to be provided. The following shows a simple policy file that gives global permission to anyone from anywhere:

```
grant {
      permission java.security.AllPermission;
};
```

This policy file can be placed anywhere on the server computer. However, its location must be specified when the *PlotterServer* is started, as will be described in Section 4.4.5.

### 4.4.3 Generate Stubs and Skeletons

We use the JDK's *rmic* compiler to generate RMI stub and skeleton from the remote interface.

First, we need to compile the remote interface:

```
javac PlotterServerInterface.java
```

This creates the class file *PlotterServerInterface.class*. Next, *rmic* compiler is applied:

```
rmic PlotterServerInterface
```

This creates the stub named *PlotterServer_stub.class*, and the skeleton called *PlotterServer_skel.class*. These two classes, as well as the *PlotterServerInterface.class* need to be copied to the *codebase* directory specified in the HTML (namely, *myClasses*).

### 4.4.4 Compile and Deploy Java Files

We may place all the Java source files we wrote, the policy file, and the Parser DLL file in a separate directory, say, c:\plotter. Next, we compiler the Java files:

```
javac *.java
```

The generated class files then need to be copied to the *codebase* directory.

### 4.4.5 Register and Start Server

Finally, we need to register and start the *PlotterServer* server. To start the RMI registry, execute:

```
start rmiregistry
```

The following command starts the server:

```
start java -Djava.rmi.server.codebase=http://hostname/myClasses/
-Djava.security.policy=c:\plotter\policy PlotterServer
```

In the above command, we need to specify the *java.rmi.server.codebase* and *java.security.policy* properties with the *codebase* directory and the location of the *policy* file respectively.

Once the registry and server are running, we can run the applet by loading it into a Web browser using the URL of http://hostname/Plotter.html.

# Chapter 5

# Conclusion

In this report, I have outlined the design and implementation of a tool for plotting mathematical functions.

**The tool is based on the client/server distributed computing model.** This means that users can share the computing resources and don't need to install the copies on their own computers. It also means that if the tool is upgraded, users will automatically receive the services from the latest version. As a result, it is very convenient for users, especially for those users who use the software only occasionally.

**The tool is simple and very easy to learn.** A concise and straightforward graphical user interface is provided to users. Users load and run the tool in a Web browser, an environment that is familiar to them.

**The tool is reliable.** It is fully tested with a variety of mathematical functions and wide ranges of variables, parameters and constants. It also handles singularities properly.

The tool is portable. All the technologies applied in this project support various platforms. Although the project was developed and tested in Windows/NT, very little efforts is needed to port it to many other platforms.

**The tool is extensible.** The tool was designed and developed based on component technology and object-oriented programming. The various components (and, within each component, the various classes) are loosely coupled, and can be upgraded independently.

**The tool is cost-effective.** All the technologies applied in this project are open and free. There is no need to purchase any special software tools. These technologies are well documented and widely used. In this way, the development cost is reduced to a minimum.

**The report has also demonstrated how to reuse a legacy system written in a different programming language.** The reuse of the parser component has further reduced the development and testing cost. It has also allowed us to adopt different programming languages for different specialized purposes. For this tool, we have used Java for Internet programming and C++ for parsing expressions. In this way, we can benefit from their distinct features.

Although it is a relatively simple application, **this tool integrates many cutting edge technologies**, such as RMI, JNI, JFC(Swing), and Java Plug-in. It shows that these technologies can work together smoothly. It is a very useful prototype system either for a more sophisticated client/server plotting software or for a similar system embedding these technologies.

From this work, I have gained valuable experience of analyzing, designing, coding, deploying and testing a distributed application. I have learned many exciting new technologies. It was also an excellent opportunity for me to practicing and upgrading my Java programming skills.

As for future work, this plotting tool can be extended in many ways. New functionalities can be added, for example, printing capacity, scaling and rotation capacities, font support, 3-D support, etc. Security and user authentication features can also be included. The tool can also be improved to handle large amounts of concurrent requests efficiently. It would be equally interesting to develop this application with other competing technologies (DCOM, CORBA, etc.) and compare their performance. Nevertheless, the enhancements should not sacrifice the unique feature of this tool, that is, it is remarkably simple, cheap, easy to learn, and easy to use.

# Bibliography

[1] http://www.wolfram.com. The homepage of *Mathematica* software.

[2] http://www.mathworks.com. The homepage of *MatLab* software.

[3] http://www.maplesoft.com. The homepage of *Maple* software.

[4] Anh Phong Tran, *2-D Graph Plotter: A Tool for Plotting Functions*, Master's Major Report, Concordia University, 2000.

[5] William A. Barrett, Rodney M. Bates, David A. Gustafson, John D. Couch, *Compiler Construction: Theory and Practice*, Science Research Associate Inc., 1979.

[6] Guy Eddon and Henry Eddon, *Inside Distributed COM*, Microsoft Press, 1998.

[7] Gerald Brose, Andreas Vogel, and Keith Duddy, *Java Programming with CORBA*, third Edition, OMG Press, 2000.

[8] http://www.omg.org. Object Management Group's home page.

[9] http://java.sun.com/products/jdk/rmi. Java RMI's home page.

[10] P. Emerald Chung, Yennun Huang, Shalini Yajnik, Deron Liang, Joanne C. Shih, Chung-Yih Wang, and Yi-Min Wang, *DCOM and CORBA Side by Side, Step by Step, and Layer by Layer*, http://www.cs.wustl.edu/~schmidt/submit/Paper.html

[11] Gopalan Suresh Raj, *A Detailed Comparison of CORBA, DCOM and Java/RMI*, http://www.execpc.com/~gopalan/misc/compare.html

[12] http://java.sun.com/products/jdk/1.2/docs/guide/jni/index.html. JNI's home page from *Sun Microsystem Inc.*

[13] http://www.javasoft.com/docs/books/tutorial/uiswing, JFC/Swing tutorial from *Sun Microsystem Inc.*

[14] http://java.sun.com/sfaq, Java Security FAQ from *Sun Microsystems Inc.*

[15] http://www.javasoft.com/products/plugin. Java Plug-in's page *from Sun Micro-systems Inc.*

# APPENDIX

# Selected Pieces of Code

## A.1 The Java code for JNI

The following shows the code in the *ParserInterf.java* (see section 4.3.2 for description). It defines the native methods that can be called by the *PlotterServer* class.

```
class ParsInterf {
        public native int ParseExpr(String jexpr);
        public native int SetVariable(String jname, double value);
        public native double EvalExpr()
              throws IllegalArgumentException;
        public native void ClearExpr();
        static {
              System.loadLibrary("parser");
        }
        public int parseExpr(String expr) {
              return ParseExpr(expr);
        }
        public int setVariable(String nm, double val){
              return SetVariable(nm, val);
        }
        public double evalExpr() {
              return EvalExpr();
        }
        public void clearExpr(){

              ClearExpr();

        }

  }
```

## A.2 C++ code for JNI

This section lists the code in the *ParsInterfImp.cpp* (see section 4.3.2 for description). It connects the Java native methods shown in A.1 to the reused parser module.

```
#include <jni.h>

#include "ParsInterf.h"

#include "parseval.h"

#include <stdio.h>

#include <iostream.h>

#include "string.h"

static Parseval* p = NULL; // a pointer to the parser object

// ifValid is a flag to indicate if the current expr is valid

bool ifValid = false;

// The method call parser's parse method to parse an expression

JNIEXPORT jint JNICALL

Java_ParsInterf_ParseExpr(JNIEnv *env, jobject obj, jstring jexpr){

   if (p == NULL)

         p = new Parseval();

   const char* expr;

   expr = (env)->GetStringUTFChars(jexpr, NULL);

   if (expr == NULL)

         return 0;

   char ex[100];

   strcpy(ex,expr);
   // Try to parse the expression.
   p->parse(ex, ifValid);
   env->ReleaseStringUTFChars(jexpr, expr);
   return ifValid?1:0;
}
// This method call parser's set method to define a name
JNIEXPORT jint JNICALL
Java_ParsInterf_SetVariable(JNIEnv *env, jobject obj, jstring jname,
                            jdouble value) {
   if (ifValid) {
         const char* name;
         name = (env)->GetStringUTFChars(jname, NULL);
         char nm[100];
         strcpy(nm,name);
         // Set variable 'nm' to 'value'.
         if (p->set(nm, value)) {
               env->ReleaseStringUTFChars(jname, name);
               return 1;   // successful
         } else {
               env->ReleaseStringUTFChars(jname, name);
               return 0;   // the variable name is not found
         }
   } else {
         return -1;  // no expr is defined.
   }
}
// This method call parser's eval method to evaluate an expression
JNIEXPORT jdouble JNICALL
Java_ParsInterf_EvalExpr(JNIEnv *env, jobject obj) {
   if (ifValid) {
         double val;
         // Evaluate the current expression.
         try {
               val = p->eval();
               return val;
         }
         catch (Error *perr) {
               char msg[200];
               char* str = perr->getMsg();
               strncpy(msg, str, 199);
               delete str;
               jclass cls = env->FindClass
                            ("java/lang/IllegalArgumentException");
               if (cls != NULL)
                    env->ThrowNew(cls, msg);

               /* free the local ref */
               env->DeleteLocalRef(cls);
          }
   } else {
         return 0.0;
   }
}
// This method deletes the current parser
JNIEXPORT void JNICALL
Java_ParsInterf_ClearExpr(JNIEnv *env, jobject obj) {
    if (p) {
         delete p;
```

```
            p = NULL;
        }
        ifValid = false;
}
```

## A.3 Converted HTML File

The following shows the HTML file converted by the Sun's HTML Converter. It is compatible with Java 1.2. (see section 4.4.1 for description.)

```
<!--"CONVERTED_APPLET"-->
<!-- CONVERTER VERSION 1.0 -->
<OBJECT classid="clsid:8AD9C840-044E-11D1-B3E9-00805F499D93"
WIDTH = 400 HEIGHT = 400
codebase="http://java.sun.com/products/plugin/1.2/jinstall-12-
win32.cab#Version=1,2,0,0">
<PARAM NAME = CODE VALUE = "PlotterClient.class" >
<PARAM NAME = CODEBASE VALUE = "myClasses/" >
<PARAM NAME="type" VALUE="application/x-java-applet;version=1.2">
<COMMENT>
<EMBED type="application/x-java-applet;version=1.2" java_CODE =
"PlotterClient.class" java_CODEBASE = "myClasses/"
WIDTH = 400 HEIGHT = 400
pluginspage="http://java.sun.com/products/plugin/1.2/plugin-
install.html"><NOEMBED></COMMENT>
</NOEMBED></EMBED>
</OBJECT>
<!--
<APPLET  CODE = "PlotterClient.class" CODEBASE = "myClasses/"
```

```
WIDTH = 400 HEIGHT = 400 >

</APPLET>

-->

<!--"END_CONVERTED_APPLET"-->
```